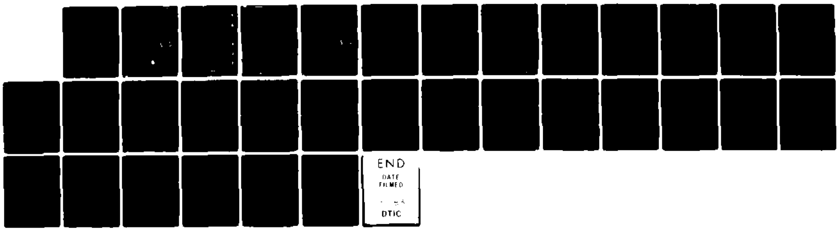
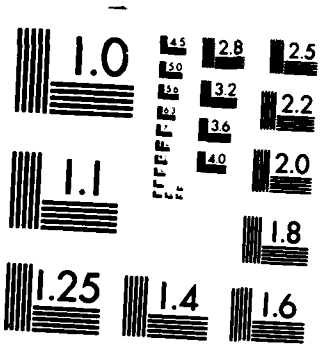
AD-A125 013 THEORETICAL ISSUES IN SOFTWARE ENGINEERING(U) PRATT AND WHITNEY AIRCRAFT GROUP WEST PALM BEACH FL D HAMLET SEP 82 AFOSR-TR-83-0038 F49620-80-C-0004 1/1

UNCLASSIFIED F/G 9/2 NL



END
DATE
FILMED
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF MELBOURNE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0038	2. GOVT ACCESSION NO. AD-A125013	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THEORETICAL ISSUES IN SOFTWARE ENGINEERING		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dick Hamlet*		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0004
9. PERFORMING ORGANIZATION NAME AND ADDRESS United Technologies Corp. Pratt & Whitney Aircraft Group W. Palm Beach, Florida		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE September 1982
		13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The discipline of software engineering has transferred the common-sense methods of good programming and management to large software projects. It has been less successful in acquiring a solid theoretical foundation for these methods. The software development process has been divided into phases, each separating itself from programming some aspect for independent consideration. The division itself has no justification save practice that has evolved for large, concurrently processed programs. Furthermore, each phase needs formal description and analysis. The author briefly describes the discipline, gives its (CONT.)

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONTINUED: existing theoretical basis, and lists some problems that need theoretical work.

THEORETICAL ISSUES IN SOFTWARE ENGINEERING

Dick Hamlet+

Technical Report 82/8
September, 1982

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia

a

Abstract

The discipline of software engineering has transferred the common-sense methods of good programming and management to large software projects. It has been less successful in acquiring a solid theoretical foundation for these methods. The software development process has been divided into phases, each separating from programming some aspect for independent consideration. The division itself has no justification save practice that has evolved for large, concurrently processed programs. Furthermore, each phase needs formal description and analysis. ~~We~~ briefly describes the discipline, gives its existing theoretical basis, and lists some problems that need theoretical work.

Docum[illegible]

+On leave from Department of Computer Science, University of Maryland, College Park, MD 20742, USA. This work was partially supported by the United States Air Force Office of Scientific Research under grant F49620-80-C-0004.

A preliminary version of this report was presented at the Winter School in Theoretical Computer Science, University of Queensland, St. Lucia, July, 1982.

1. Software Engineering

"Software engineering" is the name given to the art of programming (and surrounding activities) when art is to be replaced by a discipline using well defined methods and formal skills. Software engineering is perhaps no more than ten years old, and like most new disciplines its record is mixed.

On the one hand, a great deal was learned about good programming practice in the mid-1960s, and the spread of this knowledge beyond expert programmers can be credited to software engineering. Since the first computers were built, the problems set for them have required program solutions of great complexity. Managing this complexity requires method and discipline, and software engineering has supplied these, often from common sense or clever-programmer origins. What is undeniable is that the large system projects of today (say a real-time system of half a million lines of high-level code, created by 50 people over several years) could not be completed without so-called "modern programming practices." The transfer of knowledge to routinely trained technicians, the codification of common sense, and the introduction of management control are certainly functions proper to engineering, and software engineering has done these things for programming.

The success and growth of an engineering discipline has never rested entirely on organization of trial-and-error knowledge, however. Application of deep theoretical results is also required to progress beyond the initial success that spreading common sense brings. Historical examples in electrical engineering are particularly revealing. Early electrical experimenters are unmatched (Faraday and Edison come immediately to mind), yet their skill could not cope with the problems they themselves discovered. After a certain point, electrical engineering could progress only by applying deep results from physics and mathematics. The role of the engineer is sometimes to invent the required theory; more often it is only to apply an idea from a more abstract discipline to a problem the engineer knows intimately, where he is able to see the application as the theoretician does not. Furthermore, the application must meet a requirement peculiar to engineering: it must be in a form that can be routinely used to solve practical problems. Here Heavyside's use of "complex frequency" transformations to solve circuit problems is an example that may remain forever unmatched. The theory is difficult, but with prepared tables and undergraduate training, most people can solve problems that would have defeated Maxwell.

1.1 Role of Mathematics

Software engineering has not done very well in finding and applying theoretical work to its problems. The fault has two obvious causes: (1) a new discipline takes time to develop; (2) software is so much a creation of the human mind that no hard-science results seem to apply. Only the second point deserves comment. What other, more abstract disciplines underlie software engineering? Certainly mathematics, but not physical science. Psychology and sociology play a role, but they themselves look to mathematics for theoretical backing. The mathematical fields most relevant to software engineering are logic, algebra, and probability. Three models for creating the appropriate theoretical foundations may be described as follows:

(Prevention) Everyone should know quite a lot of conventional mathematics, because it trains the mind and perhaps it may be applicable later. No attempt to provide applications, or prune the theory to the essential is appropriate in this view. It is often the basis for university computer-science courses.

(Pure) Mathematical theories have a life of their own, and some areas are obviously relevant to computer science, so work in those areas should be undertaken. When questions arise about what is to be pursued, the choice should be made on conventional mathematical grounds of depth and elegance. The deepest results come from people holding this view.

(Applied) Starting with a problem and using whatever mathematical ideas are appropriate, a solution to the problem in abstract terms is sought. The applied approach differs from the pure in that decisions about direction are dictated by the problem to be solved, not the quality of the mathematics. As a result, the mathematics is often shallow.

If there is a bias in this paper, it is in favor of the applied view, and against the prevention view.

1.2 The Software Life Cycle

The subject of software engineering is defined by the so-called life cycle of a program [1]. The division into phases of a cycle is intended to divide the programming problem so that the parts have distinct goals. (This division is not universally accepted--see Section 1.3 for an alternative view.) Associated with each phase is a tangible "product." The existence of products is important because it provides a focus for discussion of a phase, and the product is often a subject of research. The phases are: Requirements, Specification, Design, Coding, Testing, and Maintenance. We give brief descriptions and a running example:

Requirements [2]

A problem to be solved by programming begins in the minds of people as an idea that something is to be done using a computer. These people pose the problem, but intend to have others solve it. They are presumed to be expert in the problem subject, but not necessarily in computing. Furthermore, since the solution has yet to be realized, its form may be less clear in their minds than is the problem. The statement of requirements is expressed in English or some other natural language, making use of technical terms in the problem area. It is precise in describing the problem context, less precise in isolating the problem, and perhaps very fuzzy about the required solution.

For example, consider the problem of automatic inventory control for a manufacturer. The managers can describe their inventory very well: it consists of such-and-such materials, stored in such-and-such locations, transported in this way, obtained from certain sources of supply with certain delays, and utilized in a process that is understood in exact detail. The inventory problem is that unless care is taken with every factor, materials may run short or long, causing expensive disruptions. (The problem definition is less precise than that of the inventory itself--hard definitions are lacking.) Computers manage information, so one might be used to solve this problem, say by preparing timely reports for the attention of those who obtain and use materials. Just what these reports should be, and what constitutes "timely" are vague. The managers can judge the success of a computer solution--it will or will not eliminate the disruptions that are believed to be caused by poor inventory control--but they cannot capture the character of a successful solution.

The situation in requirements is aptly described by the aphorism "the end user knows what he has, but not what he wants." It is in the nature of requirements to change as proposed problem definitions and solutions become available.

Specifications [3]

Specifications can be thought of as requirements made precise as to the existing problem and a proposed solution. More important than the precision is the change in viewpoint: a specification is done by those who intend to solve the problem, as the first step in that solution. Specifications are truly separate from the phases that follow only if they are restricted to describing what the problem and solution are, not how they come about. Because people seem to think best about complex situations in terms of actions, sticking to the "what" can be very difficult.

Formal languages enter the picture at the time of specification. These languages are sometimes described as "non-procedural" or "declarative" to indicate that they do not give algorithms, but merely describe properties of solutions. One way to abstract and sharpen the idea of specification is to limit it to "input-output behavior." Thus inputs which will be presented to the computer program are described, each with its corresponding desired output.

In the example of inventory control, a specification might define the required inputs as daily transactions, listing each item bought, sold, or moved, and further define a "state" of values of all possible items against which these transactions are to be made. Here the formal language required is perhaps no more than set theory, where the material items are the set members. Secondly, the detailed format of each report to be prepared, and the sequence of issue relative to the occurrence of transactions, would be defined. Here a more complex descriptive mechanism, perhaps that of the context-free generative grammar, might be required. The desired relationship between the input transactions (in some initial state) and the resulting reports might be described by first-order predicate logic: assertions could be written in variables representing the values of material items, to relate the transaction values to those appearing in the reports, and to characterize as predicates situations in which a report is required.

Design [4,5]

The design phase begins to supply the "how" of the problem solution. Its primary purpose is to divide the solution into pieces, each of which can be realized as a self-contained program unit. (Although the word is much misused, these pieces of program are called "modules.") Dividing the problem in this way has the twofold purpose of intellectual control, and the utilization of many programmers simultaneously. Decisions about the programming language(s) to be used, the computers involved, and the basic organization of the program (for example whether it will run as a unit, in a series of overlays, or as a collection of cooperating processes) must be made before the division into modules.

Formal design languages (the abbreviation "PDL" is used, the "P" for "problem" or "program") are often like conventional procedural programming languages except that data structures and actions within programs are allowed to be commentary. PDLs may thus be thought of as poorly defined high-level programming languages. (However, insofar as a PDL is precisely defined, its use may overlap with the coding phase of the life cycle to follow, and be unsatisfactory in separating problem division from the piecewise implementation.) Other design aids are usually graphical in nature, including multilevel flowcharts, module-relationship diagrams, etc.

In the inventory-control example, the design might be for a batch comput-

[illegible] database defining the state of the inventory (the actual disk-resident structures are part of the design, perhaps in a special language). The transaction-processing program would update the database, and might be divided into an input-classifying module, an updating module, and utilities to manipulate the files. Each of these might be described in a series of flowcharts. Each report would be prepared by a separate program, these programs to be scheduled at the completion of each transaction cycle depending on the state of the database. Report generators would share utility modules with the transaction program, but their main components would be formatting modules peculiar to each report.

Coding

The module descriptions of the design, along with detailed interface information, are in a form suitable for distribution to programming teams. The programmers need not understand requirements, specifications, nor overall design, but only realize limited objectives module-by-module.

All the traditional formalism of programming enters the life cycle in the coding phase. Conventional, efficient programming languages are processed by conventional compilers.

The inventory control modules might be written in assembler for the file utilities, in COBOL for the input classification module, and in a report-generation language. These program units would be linked together to form the transaction and report programs.

Testing [6]

Software testing can be a complicated phase with many subphases, if the software itself is complex. In the simplest case testing is divided into "unit" and "integration" parts. Unit testing involves only single modules, and is typically done by the programmers to convince themselves that each module performs as it should. Integration testing is conducted on collections of modules, often complete programs. Of course, only running programs can be tested at all, so part of the testing process may be the construction of "driver" and "stub" modules that allow incomplete collections to be executed. Unit testing depends on the peculiarities of each module, and cannot be planned for out of the serial sequence of the software life cycle; however, an integration test can be devised in detail as soon as the specifications are available. Integration "test plans" are therefore sometimes begun in parallel with the specification phase. Where a software system is extremely complex, its integration test may be divided into phases called "functional" (in which various requirements are tried to see if they work as specified), and "system" (in which stress is applied to the system in the form of extreme values and loadings, in which variations in target-computer configurations are tried, etc.).

The early parts of a program test have a lot in common with debugging, and support systems have grown up for each language. However, in the later stages of testing, the tester is seeking not errors, but confidence that when no errors are found it means that none exist. The only known scheme for obtaining this confidence is to "exercise" the program code, making sure that each statement, expression, control branch, etc., is used in some test.

Following unit test of the modules of the imagined inventory-control program (which would require a simulation of the database utilities as stubs), the various programs can be tried, for example by generating a report with a certain database, then applying simple transactions, and repeating the reports, which should show the appropriate changes.

Maintenance [7]

As soon as a software system is delivered to its users, it requires changes, for two reasons: it contains bugs, that is, it does not do what it claims to; and, it has been constructed to do the wrong thing. Bugs are an inevitable consequence of the complexity of computer-handled problems and their programs; failures of intent arise naturally from imprecise requirements: the end user did not know what he wanted, but experimentation with the finished program quickly shows it was not that. Bugs might be thought to enter only in design and coding, but specifications and even requirements can be at fault--they can call for inconsistent results that cannot be realized; of course, testing has failed to uncover any bugs that remain on delivery. Solving the wrong problem evidently calls for beginning again. However, in practice, the product of the entire cycle is the executable code, and it is common practice to change only that. The result is that requirements, specifications, design, and test plans are rendered useless by code changes that they do not reflect. These products were thought necessary to create the code in the first place, but with the code in hand, they are frequently discarded.

It is all too easy to imagine the dissatisfaction of a manufacturer with the delivery of the batch transaction system of our running example, when he discovers that batch processing introduces too much time delay and the reports fail to prevent inventory shortages. The managers then realize that what they really need is an on-line system in which all materials changes are entered as they take place, and exception reports generated immediately. No sooner would such a system be delivered than they discover that a real-time inquiry capability is essential, and so on.

1.3 An Alternate View of Software Engineering

The software life cycle was created by splitting phases from coding wherever possible. For example, the design phase isolates the division of code into modules. The product of each phase is created from the information contained in earlier phases (in strict sequence with the exception of some test plans), but is to be evaluated using its own internal criteria. For example, specification relies on requirements for information, but calls for precision and a non-operational character. Coding is thus singled out as primary. It has a natural existence that all other phases lack, it has the most precise, well developed product, and the compromises of practical maintenance show that it alone matters when push comes to shove.

There is a dissenting view of software engineering which has its source in the primality of code. In this view, the splitting of software development into phases introduces more problems than it solves. In particular, the very different products of each phase and the different formalisms thus called into being, seem inferior to a single product and a uniform formal treatment. In this view what is needed is a language which has the precision of a conventional programming language, in particular which can be mechanically executed, but whose character makes it appropriate for specifications or even requirements. The argument is that were such a language used, there would be no need to split off concerns around coding: the code would be its own specification, correct by definition, requiring no test. The problem of users not knowing what they want would remain, but the cycle of adaptation to change would be shortened, and the product maintained would be the one developed. It would still be possible to separate phases for study and development, but this would be done by processing the single code product in different ways.

The alternate view is attractive, its sole drawback being that with it a software problem must be solved in a single thrust, not broken into five parts for solution. The intellectual economy realized, however, might pay off for medium-scale problems. (Small-scale problems seem to require nothing more than programming--any method seems to work in competent hands.) This paper takes the conventional life-cycle view, but tries to discuss the issues of a single software-engineering language under "specification" in Section 2.3.

2. Theoretical Issues

Research in software engineering has naturally focused on the product X of each phase, to explore how X can be improved. This focus is successful partly because the phases have distinct goals. Even the dissenting view that the life cycle should be collapsed to a single programming phase can be fitted into this mould, by adding to the specification phase the requirement that its language must be compiled and executed.

2.1 Programming Language Theory

The model of a proper relationship between theory and practice occurs in the coding phase of the software life cycle. Here the product is a computer program, and the formal abstraction is the programming language from which it is drawn. Programming languages have their origin in practice and experimentation. For example, the variants of IAL developed in the 1950s are in most ways the equal of any language in existence today, and they came directly from the idea of the stored program and some simple mathematical analogies. (Indeed there exists a tradition of devising programming languages without any reference to existing mathematics--reinventing formalism as needed--that produced ALGOL 60 and ALGOL 68.) The role of theory in programming languages has been to explain and clarify rather than to prescribe language features.

For syntax, theory has been spectacularly successful. Regular expressions and context-free grammars are marvelous devices for describing and understanding sets of strings over an alphabet, and they are furthermore easy to teach and learn. (It is worth remembering how difficult the ALGOL 60 syntax was considered when it first appeared, but this parallel can be overdrawn.) Semantics of programming languages is more difficult than syntax, but there has been some success here as well. The ALGOL 60 report [8] is unmatched in its use of English to define programming language meaning (using two clever devices: equivalent programs and the copy rule), but the need for formal treatment is apparent. Examples exist of both the pure and applied mathematical approaches [9]. Operational semantics is applied. The formalism is shallow and closely follows the practical problem of language implementation. It is undeniably useful to the compiler writer, and in the understanding of arcane language details, but unsatisfying as mathematics. Denotational semantics combines the pure and "prevention" approaches. Its presentation makes no attempt to avoid conventional mathematics that might be a barrier to the uninitiated--for example lattice theory--and the theory often fails to illuminate the meaning of practical languages, but there is depth to its results.

Parsing theory [10] does not easily fit into one of the mathematical categories. Parts of it are pure theory, not directed by applications, including deep results (the relationship between the LR(1) and other languages classes, for example). Parts are no more than fitting established compiler-writing techniques into a formal framework (syntax-directed translation, for example). And part seems simply unsuccessful (the LL(k) explanation of recursive descent with lookahead, for example). It is unclear whether the compiler-compilers based on table-driven models are an improvement over those that arose from practice alone.

The problems of programming languages in the development of software are ones of power and control. On the one hand, language features fitted to handle difficult problems with ease are desirable; on the other, such features in unexpected combination often lead to programs that do not work and cannot be easily repaired. Language theory has not contributed much to solving these problems, nor is it likely to do so. There have been some attempts to equate the ease or clarity of theoretical description with language quality, but these are probably misguided. Some features of syntax are inherently non-regular (parenthesis balancing, symmetric statement nesting); others are inherently non-context-free (matching of declaration and usage forms). That these ideas are difficult to describe in otherwise-satisfying formalisms may be relevant to philosophy, or a comment on the formalisms, but has little significance for language design. Similarly, some language features have difficult denotational explanations (variables, transfers, exception handling); but, these features may still be useful and important.

2.2 Software Engineering Problems--Outer Phases

Because programming itself is the best understood part of the software development cycle, yet farthest from software users, the outstanding problems of software engineering grow less precise and more important as they occur in phases more widely separated from coding. The problems of design and testing are technical and difficult, but their complete solution would be less important than a beginning maintenance or requirements/specification theory, where technical precision is lacking.

Requirements/specification interface.

Between the requirements phase and the specification phase, people who have a problem turn it over to those who hope to solve it. When they do so, the communication is less than perfect, and because software professionals work with formal, technical ideas, there is little feedback after the turnover. The software system that develops may be wrong from the outset, but the people who requested it have no way of knowing this until it is delivered. The requirements/specification problem is thus one of finding a way to keep the nontechnical user informed as the technical development proceeds. Perhaps this is largely a problem in psychology or sociology, but it has some technical aspects. The only reason that the software user knows of failure is by trial. It has therefore been proposed that a means of "rapid prototyping" [11] or simulation be employed throughout the development process. As each decision is made, it should be communicated to the user in the form of a system that can be tried. These simulated or prototype systems must be cheap and easy to produce, and they must accurately reflect the technical development, but their efficient use of computer resources is less important.

Rapid prototyping raises two recurrent themes in software engineering. The first theme is that of language execution by machine. If a specification is "executable," particularly in a partially completed state, it can support feedback to the user. The second theme is that of finite support for infinite objects. When a user tries a prototype or delivered system, he necessarily chooses only a few of the many cases that the system might process. If these cases are "representative" they give an accurate picture; but, how can good cases be selected? The same question arises during the test phase of development. Thus the apparently vague but important questions of how to improve user/technician communication can be translated into technical questions in the specification and testing phases.

Maintenance

A second important software engineering problem occurs at the end of the cycle. Delivered software usually fails to meet requirements. Its deficiencies range from simple malfunctions that escaped detection to "features" that are just what the users do not need, or inability to handle parts of the problem at all. The conventional view of software maintenance is that it is a development cycle in miniature, including all the phases from requirements analysis to testing. This view ignores the primary fact about practical maintenance: it is supposed to be far faster and cheaper than repeating the full development process. Because a complex system has a complex specification, design, etc., even small changes cannot be introduced without extensive study of the original, which may be literally more difficult that recreating it from scratch. (A second important characteristic of real maintenance is that it is seldom undertaken by the people responsible for original development.)

The division of software into modules is the key to maintenance [12], if the interconnection allows some to be changed without understanding the others, and if changes involve only a few modules. Once a proper decomposition of the system has been made, it is evident that the intermodule independence must be maintained, and there are good techniques for mechanically controlling the extraneous linking of modules, notably the principle of "information hiding" as implemented by an abstract data type. However, there is no theory to guide the original division, and information hiding is actually counterproductive when the modules divide the problem badly. If changes can be anticipated, the designer can think about them in modularizing the system; what is needed is a theory of change to deal with the more common case of unexpected changes.

An ideal software system, from the point of view of maintenance, would have the properties that:

(1) The product of each phase is broken into a number of elements that can be independently analyzed; and

(2) A change within an element in one phase of the cycle results in changes confined to an element of the succeeding phase.

Sometimes this property is called "traceability;" it has received very little formal investigation. Not the least important example is that changes in a code module should not require repeating tests outside that module.

Although there is no evidence beyond wishful thinking, one might expect the problems of controlling change to be easier than those of development in general. If a system has been proved to have some property, and a change is made, a proof that the modified system retains that property should be easier than the original. One might even hope that for circumscribed changes, modifications made by hand in (say) specifications could be automatically (and correctly) reflected in the design, code, etc.

2.3 Software Engineering Problems--Specification

Formal programming languages were once only a human-computer communication device, but their usefulness for the description and discussion of algorithms soon became apparent. The benefits of a formal-language treatment are that formal objects can be analyzed, and that time spent casting an idea into precise form pays off in sharpening that idea. The goals of specification are now clear enough to permit investigation of formal specification languages. Specification "programs" should describe what a computer program is meant to do, and they should be _complete_, _consistent_, _and_ _effective_.

Completeness of a specification intuitively means that its description of what is to be done has no missing parts. The most general form of this idea involves human intent, since one person might consider it necessary to constrain the program in a way that another person would not. A particularly difficult question is: might a complete specification permit multiple output values for the same input? If a specification provides no constraint on the output associated with some input, then all results are equally acceptable; yet, it is an intuitive incompleteness that permits this to happen.

The intuitive meaning of a consistent specification is that there must not be multiple, contradictory descriptions of what is to be done under the same circumstances.

Completeness and consistency are examples of properties that might be proved for a given specification, and such proofs are themselves an important aspect of the form specifications take. A method of specification is good if it facilitates such proofs.

An effective specification is one that can serve as an oracle: it can be used to decide of any given input and proposed output, whether or not the pair is acceptable. A stronger requirement is executability, in which a specification is able to stand in for the program that meets it: the specification can "compute" output values from inputs in some way. Of course, an executable specification is effective, but not the other way around.

A better discussion of specification-phase issues is possible for particular proposed methods, of which three will now be considered: logic-based, algebraic, and operational.

Logic-based Specification Languages [13]

First-order predicate logic has the power to describe any problem that has an algorithmic solution, and may be the most natural language for the task. The form of specifications in logic follows the Floyd-Hoare correctness theory: two assertions are given, one describing assumed properties of inputs, the other describing required properties of outputs. Call these input and output assertions P and Q respectively. For simplicity, suppose that P contains exactly one free variable x (the input), and Q contains x and a distinct y (the output) free. As usual, write $P(c)$ to mean that c replaces x in P, and similarly write $Q(c,b)$. The specification itself is as follows:

> The program specified is to process any input u satisfying $P(u)$ so as to produce an output v satisfying $Q(u,v)$.

That is, the program is to be totally correct with respect to these assertions.

A specification is consistent just in case for each input satisfying P, some output satisfies Q. That is,

$$\forall x\ (P(x) \implies \exists y\ Q(x,y)).$$

The intuitive justification for this definition is that if the specification should indicate contradictory properties for the output on some input u, then no v could satisfy $Q(u,v)$.

In the most general form, a specification is complete just in case the person devising it agrees with what can be derived from it. Formally, the person should approve of each theorem of the form

$\forall x,y\ (P(x) \wedge Q(x,y) \Longrightarrow R(x,y))$

for arbitrary assertions R . The narrow requirement that there be at most one output for each acceptable input is:

$\forall x,t,u\ (P(x) \wedge Q(x,t) \wedge Q(x,u) \Longrightarrow t = u).$

The narrow completeness condition implies that there is a function f mapping objects satisfying P onto objects satisfying Q . If there is to be a program for f , it is further required that f be computable. That is, the specified program must compute f such that

$f = \{(x,y) \mid P(x) \Longrightarrow Q(y)\}.$

The consistency requirement is precisely that

$\{x \mid P(x)\} \subseteq$ domain f.

First-order logic specifications lend themselves to proofs of completeness and consistency, and other such properties that are concerned with the internal relationships of the specification. An important application is in the specification of so-called "secure" systems. Here a core of some problem is identified as sensitive, to be protected against intrusion, and the planners wish to be certain that they have thought of all possibilities that might constitute violations of this core. The necessary properties in addition to consistency and completeness have just the character that they are easily stated in first-order terms, as invariants in addition to the input-output assertions P and Q . Furthermore, such applications are considered so important that the high cost of mechanical theorem proving is acceptable [14].

A specification is effective in the strongest intuitive sense if there is an algorithm that decides the truth of both

$P(c) \Longrightarrow Q(c,d)$

and

P(c) itself

for all constants c and d . Such an algorithm would distinguish three cases: (1) P(c) does not hold, hence the input c is unacceptable; (2) P(c) holds and Q(c,d) holds, hence the pair (c,d) should be in the graph of the relation specified; and (3) P(c) holds and Q(c,d) does not, hence the pair (c,d) should not be in the graph. In a less stringent view, a specification is effective just in case an algorithm exists to decide

$P(c) \Longrightarrow Q(c,d)$

alone. In case the formula holds, the algorithm need not further decide whether this is only because of a counterfactual hypothesis. Using this definition the algorithm could certify a pair as being in the graph of the specified relation when in fact the input was unacceptable.

These alternative definitions differ when P itself is undecidable; then the specification cannot be effective according to the more stringent view, but might be according to the weaker one. There is something unreasonable about specifications for programs where the input must satisfy an undecidable proposition, perhaps because intuition says programs should be able to check for valid input. An input-checking program has an input assertion of 'true', and invalid inputs are distinguished by an error output described by Q . In this case the two definitions coincide.

No matter which notion of effective we choose, logic specifications are not in general effective, and recognition of an effective one is itself an unsolvable problem. Furthermore, it is not even possible to decide if a given specification is for a computable function. If the input assertion is reduced to 'true' the latter problem includes deciding whether or not a total function has been specified.

A logic specification is executable iff there is an algorithm for obtaining d such that Q(c,d) holds, for all c such that P(c) . Here it is evidently correct to omit the P(c) decision from the algorithm, since the specified program itself does not have to carry out the calculation of P . It seems unreasonable, however, to "execute" a specification for an invalid input by producing some arbitrary result, and this is reflected in the anomaly that a specification could be executable, yet not effective according to the more stringent definition. The best resolution of these definitional conflicts seems to be to take the more stringent definition of effective, but confine it to the case that P is decidable.

Execution of a logic specification in general requires a search of the space over which the second argument to Q ranges. Some proof techniques when applied to

$$\exists y\ (P(c) ==> Q(c,y))$$

for constant c will yield a value y satisfying the formula as a part of the proof, but they produce an arbitrary value for y in the case that P(c) is false.

In attempting to establish a special case of effectiveness with a theorem prover, that is, to prove or disprove

$$P(c) ==> Q(c,d)$$

with constant c, d , the same problem can arise, because the formulas may contain internal quantification, whose satisfaction again requires some form of search.

There is a general response to the occurrence of an unsolvable problem in practical situations--to restrict the discussion to a subset of the original situation where the problem can be solved. There is necessarily a loss of generality, because it is typical of these problems that identifying the undesirable cases is itself unsolvable. Therefore one must expect to give up a bit of baby in return for getting rid of the bath water. In fact, the only restriction attempted so far is the trivial one to finite domain. If in the output assertion Q all bound variables are restricted to finite sets, then a specification is effective, and efficiently so (say) by a proof method based on analytical tableaux. If the range of the second argument to Q is also finite, the specification is executable in the same way. The motivation for this drastic restriction may come from the application to information retrieval, where an existing data base is finite, and logic specifications are related to queries in which the quantifiers range only over the values already inserted.

Evidently more research on restrictions designed to gain effectiveness and executability of specifications is needed. A starting point for such research might be existing languages which include "features" grafted onto a first-order base [15], which are designed to gain efficiency in execution, but whose theoretical significance is unexplored. In a way such practical adjustments to logic are the worst way to give specifications: although the form appears mathematical, one cannot be sure what programs are in fact specified, except themselves. The executability they have gained is at the expense of precision.

Specification Languages Based on Algebraic Axioms [16]

Many of the "modules" into which a computer program is divided have the character of providing a service function to other parts of the software. A collection of entry points to the module are defined as parametrized operations of which it is capable; these operations map among sets of objects whose values may be manipulated in no other way. That is, the module defines certain sets and operations on those sets, and must be used for access to them. Since the late 1960s many programming languages (notably SIMULA 67, CLU [17], and Ada) have provided support for such modules, and a design idea called "information hiding" has evolved to take advantage of modules' ability to keep "secrets," releasing to the surrounding world only limited information about how the internal sets and operations are implemented. The formal objects corresponding to these encapsulating modules are called data abstractions.

Formally, a data abstraction has a syntax defined by a signature which lists the set and operation names, and gives the latter domains and ranges from the former. The semantics of this object is given by a collection of algebraic axioms, equality assertions between pairs of terms using function names of the signature, where universally quantified variables appear as atomic terms. From the logical point of view the meaning of a data abstraction can be given in the usual model-theoretic way. The set names are interpreted as intuitive sets, and operation names as functions properly mapping among those sets, according to the signature. Equality is the intuitive binary predicate within each set. Some interpretations may be models for the axioms, and in the most general view any model is a meaning for the data abstraction whose signature it matches.

Herbrand, Gödel, and Kleene used sets of equations [18] very similar to the algebraic axioms of data abstractions, as a device for capturing intuitive computability. Although the analogy has not been satisfactorily explored, it appears that data abstractions could be studied as HGK equations in which only certain natural numbers are allowed to appear, the allowed numbers being codes for values in the sets of the signature. If the coding functions are recursive the results of the theory should remain unchanged; this would restrict what data abstractions are capable of specifying, but perhaps not in practical cases. As computing devices, HGK equations utilize numerals, syntactic standins for natural numbers defined inductively starting with zero. The value of a function f (say 1-ary) whose name appears in a set of equations, given numeral argument p, is numeral q iff

$$f(p) = q$$

can be derived from the usual properties of equality, the fact that successors of equal numerals are equal, and substitution of any equation in the set with all argument variables consistently replaced by numerals.

The more common formal view of a data abstraction is as a heterogeneous type algebra of constant words formed by composing the function names of the signature beginning with 0-ary functions. The objects described are congruence classes of the word algebra that satisfy the axioms. It is usual to try to make a further restriction that selects a unique algebra from all factor algebras defined by possible congruence relations. The most commonly chosen factor algebra is the initial algebra [19], of which all others are homomorphic images. In the initial algebra, two words are equal (in one congruence class) just in case they can be proven so using the usual properties of equality and substitutivity in the axioms.

Another algebra is the final one [20]. The functions of the signature are divided into those whose range is the type being defined, and those whose range is other (supposedly already defined) types. Call the latter distinguishing functions. Two words are equal in the final algebra just in case they can be proved equal, or no distinguishing function maps them to unequal values. In many practical examples the congruence classes of the final algebra are much larger than those of the initial algebra.

A data abstraction is thus a specification for a program expressed as an encapsulated type in a language like CLU. It is possible to write implementations to these specifications without special language support, thereby giving up some information-hiding features. The type (mode) construct of Pascal (ALGOL 68) suffices.

Any data-abstraction specification is consistent, because in the usual algebraic way, "inconsistencies" simply mean that objects which appeared to be distinct are not. Adding a "contradictory" axiom to an existing list can only widen the congruence classes, making more words equal. In the extreme case, a singleton set whose element is mapped to itself by all functions is a model. Similarly, the unique initial and final algebras always exist. Thus the most "contradictory" set of axioms can at worst lead to a meaning more trivial than intended. Hence the only kind of consistency involves a person's judgement that the objects of the specified type are congruence classes of exactly the right size.

The addition of axioms to a signature may permit equality proofs impossible without the added axioms. Since a person might consider a specification intuitively incomplete if words expected to be the same are not provably so, completeness can be defined as a human being's agreement with the totality of all equality proofs. The narrow idea that two results should not be possible for the same input has no meaning for data abstractions. If no two words are equal then there are no axioms, or only trivial ones; when there are nontrivial axioms, some constant words will be equal, but their different appearance does not make them "different results."

A data-abstraction specification can be effective in several different senses. From the algebraic viewpoint, effectiveness amounts to the ability to transform one name for an abstract object, the one in which each function application explicitly appears, into another, simpler form. The desired canonical forms may be those in which application of the axioms has eliminated redundant or cancelling function names; or, the canonical words may be required to use only a restricted set of function names, subject to further constraints. For example, in a signature for natural numbers, the canonical form might be a sequence of applications of the successor function to zero. Then the specification would be effective if there were an algorithm to answer all questions like:

Is $(0' + 0')' = 0'''$?

Because questions of equality in both the initial and final algebras are decided by the existence of proofs, data-abstraction specifications are not in general effective. For some canonical forms and some axioms, it can happen that a word is equal to no word in canonical form. In this case, the specification is usually called incomplete.

Similarly, "executability" from the algebraic viewpoint is the existence of an algorithm to produce a canonical form for an arbitrary word. Thus a specification is necessarily effective if is executable, and if it is incomplete it can be neither.

There is an attractive candidate for a general algorithm to execute data-abstraction specifications: use the axioms as rewriting rules. Several computer systems to aid in designing specifications use this algorithm [21, 22], and have therefore generated interest in characteristic conditions for its success, to identify those specifications on which the systems can be used.

The algebraic form of effectiveness ignores an abstraction's implemented representation, however. In an implementation, the objects involved are not constant words, but patterns of bits thought of as representing numbers, strings, radar signals, etc. The questions of effectiveness and executability of a specification involve these representations. For example, if a signature set S is strings, and an operation m maps S into S, while E is a zero-ary map, then it is uninteresting that m applied to $m(m(E))$ is (say)

$$m(m(m(E))) = m(E)$$

in a canonical form where the least number of m operations appears. Rather, one wants to know the result of applying m to (say) 'HELLO'.

The representation is not part of the specification, yet the ability of the specification to act as an oracle depends on it, because the inputs and outputs must be put into correspondence with the constant words. Because implementations are done from intuitive understanding of the abstraction, and because representations are seldom unique, the correspondence is onto an intuitive model, and is many-one. The interpretation of the words into the same model is then required to permit execution, and this mapping may only be given for canonical words. In these circumstances, the effectiveness of the specification depends on these mappings. In the simplest example, let there be one set S and operation $m: S \rightarrow S$. Let the mapping from implementation objects to a model be R, and from canonical constant words to the model be a bijection I. The specification is effective if for any implementation object x and y, it is possible to tell if

$$m(I^{-1}(R(x))) = I^{-1}(R(y)),$$

a question that can be answered by a proof in the word algebra when $I^{-1} \circ R$ is effective. (The form

$$I(m(I^{-1}(R(x)))) = R(y)$$

is less useful because the equality must be decided in the model, where proofs are intuitive.) The specification is executable if for any x it is possible to find a y satisfying the formulas. The remarks about exhaustive search for logic specifications thus apply with equal force to data-abstraction specifications, but the latter are often effective.

The formalization of the mappings between implementation and abstraction, and proper placement of these in one world or the other are theoretical problems in data abstraction. Another problem concerns the mismatch between these worlds. The word algebras seem to capture exactly the sort of sets and operations in which invoking a module leaves no trace beyond the value returned. That is, there is no persistent internal storage, and the parameters are called by value. When perfectly natural (from the programming viewpoint) extensions are made to relax these restrictions, the algebras no longer correspond to the implementations. One might expect that this situation could be explored to find reasons why constructions such as call-by-reference are unnatural from the algebraic viewpoint, or find algebraic models for them.

Operational Specification Languages [23]

Both data-abstraction and logic specifications have strong mathematical roots, which may explain their strength on consistency and completeness (and their weakness on executability). Operational specifications on the other hand arise out of programming itself, and stress execution.

Specification was to separate from coding "what" is to be done, avoiding "how." If a program is the specification it certainly goes well beyond "what." The explanation offered by proponents of operational specification is that their programs are natural for people solving problems, but inefficient for computer implementation. Thus "very high level" specification languages are an appropriate step prior to coding. Furthermore, it is argued, this prior step is not design, since the operational specification is a "how," but not one that can be actually used in the program.

Operational specification languages have many features based on experience with the problem domains for which they are designed (and most are designed for large, concurrently processed, real-time systems), and on tricks learned from the psychology of conventional programming. Only a few of these features have theoretical significance. We single out: (1) cooperation among processes, (2) stepwise refinement.

The idea of a process arose in multiprogramming operating systems, where several simultaneously active tasks had to communicate in a controlled way. So-called "cooperating sequential processes" [24] then developed into a programming technique in their own right, particularly for real-time systems where independent routines are needed to handle time-critical functions, and in multiprocessor systems, to take advantage of the possibility of concurrent execution. A process is most easily imagined as a complete program, whose execution is not quite self-contained. At some points its computation is connected with the computations of other programs, and the ensemble behavior is crucial to the intended operation. The collection of processes that make up a system can of course be described without reference to this internal structure, since the behavior of the system transforms input to output in the usual way. But if the inputs and outputs can be naturally grouped in some way, it makes sense to separate the program into parts that handle each group, and these are the processes of an operational specification. The connections between processes take many forms; the most common is a message-passing facility with the ability to test for (or await in some way) an incoming message. Some operational specification languages impose a further structure on processes, insisting that (except for the communication) they be very simple, perhaps finite-state-like programs, often required to operate in an indefinite repetitive cycle. The power lost within each process is gained back through their communication.

Step-wise refinement is a programming technique invented in the early 1960s, and more recently given academic respectability [25]. In it a program is organized into procedures so that its operation can be understood in a series of levels. At the top level there is a single procedure, whose body uses a few complex data types and very little control logic, calling on other procedures when the necessary processing becomes complex. These procedures in turn use simpler data types and invoke yet other procedures, until at the bottom level the procedures use only basic data types, and invoke no other procedures. An understanding can be gained at any level by ignoring procedures at lower levels, taking calls on them to have the effect suggested by commentary at the upper level, and taking data types there as properly implemented elsewhere. (The role of data structures in this technique is less well understood than the role of procedures, since the type-defining mechanisms are much more recent.)

Some operational specification languages [26, 23] use step-wise refinement ideas so that the specification is given only at the upper levels. Thus some procedures and data types are left formally undefined. This is part of a specification method that parallels the program-development method, but it has implications for executability as well. When the specification has been worked out only at a high level, it can still be tested if values can be supplied for the missing objects. People can do this, or if the sets from which values should come are known, random selection can be used to generate missing values. Using a specification language in this partially defined way makes it useful for the requirements phase, since the technical decisions being taken at each level can be communicated to the user in the form of tests. In a sense, these test executions are new objects deserving some formal study, because they are examples of program meaning that mirror its static structure. At the top level the processing is precise, while at the bottom it is fuzzy.

If the programs of operational specification languages had the same properties as conventional programs, their specifications would be consistent in the sense that a well-defined semantics for programs gives a functional meaning. Such specifications would be effective (indeed, executable) in the strong sense that the language definition provides a means of working out the output from any input, subject to the problem of halting. Hence the effectiveness would be uniform over all specifications. Completeness has to do with the domain of the specification program: if that domain agrees with what a person intended, the specification is complete. Thus if specification programs were conventional, only difficulties with halting could keep them from being consistent, complete, and executable. The introduction of concurrent cooperation changes this picture completely, but to discuss the more complex situation we must take a particular operational specification language.

As a straw-man language, let us imagine that each process is a finite-state transducer, modified so as to communicate with other processes. Add a new class of states whose symbols come not from the input, but from another transducer, and yet another state class whose outputs go to another transducer. This communication takes place as follows: any interprocess output produced is appended to a communications string, and interprocess inputs are obtained from the beginning of this string. The difference between a collection of such transducers and conventional ones is that the input (and the communications string) are thought of as dynamic objects. Should any process call for input when the necessary string is empty, that process simply waits until there is a symbol available. Should several processes be waiting, any of them may consume a newly available symbol.

This language isn't very like actual specification languages, but it exhibits all the features we want to discuss. In it, programs can be written so that they cooperate in a perfectly deterministic way, synchronized so that the pattern of which machine is active is controlled at all times. It is also possible to write programs that "deadlock." Deadlock is defined as a situation in which all processes are in an interprocess input state, and the communication string is empty. This situation will then persist. The cases in between are the ones of interest: the pattern of machine activation has many possibilities, but none of these lead to deadlock, and the overall input-output behavior of the system is the specification. In the sequel, "operational specification" will refer to programs written in this language.

Such operational specifications are no longer necessarily consistent. It can happen that several processes compete for an input or communications-string symbol, and depending on which obtains it, the input-output results differ. (In the jargon, the programs are "non-deterministic.") It seems reasonable to define consistency of a specification as the impossibility of this happening; that is, for each input, the same output must appear no matter which choices are made about dispersing input or communications-string symbols, and no matter at what speed the input string appears relative to processing. For a given input, consistency can be checked by trying all the sequences, but in practice the task is combinatorically intractable, and proof methods for concurrent systems are not well developed. Properties of operational specifications are usually tested instead of proved, but since a test may fail to investigate many of the potential activation-sequence alternatives, its result may be misleading. Indeed, the possibility of an inconsistent specification throws doubt on effectiveness itself: when the specification serves as an oracle, it may be delivering only one of several possible results.

A specification that deadlocked on some but not all activation sequences would be inconsistent; but if all sequences deadlock for some input it is still deficient, and perhaps it is appropriate to call it "incomplete." The domain notion of completeness is inappropriate, because it is common for invalid inputs to be initial sequences of other, valid inputs, and there is no way to cause the specification to be "undefined" on the former and not the latter.

Successful analysis of operational specifications depends on understanding nondeterminism. In this the relationship between individual-process behavior and whole-system behavior seems to be crucial: sufficiently drastic local constraints might make the global analysis possible. However, there is a paradox here: if processes with simple behavior are combined, the composite behavior will be unlike its components [27]. The arbitrary combination of processes does not create another process, but something more complex. Yet the fundamental analysis technique is decomposition of a structure into substructures that share its properties. Perhaps it will be necessary to have two communication mechanisms, one that preserves severe restrictions on process behavior and a more general mechanism, the latter to be used only at the top level.

2.4 Software Engineering Problems--Testing

A requirement that specifications (or requirements) be executable raises the testing problem even in the early phases of software development; certainly it occurs in design, coding, and maintenance. That problem is intuitively the following: since tests are by their nature mechanical, involving computers rather than people, how can the (small) finite number of tests it is possible to conduct be assessed as predictions about all future usage of the software? In requirements this question takes the form of the significance of the few cases a user tries to convince himself that a specification is what he wants; in operational specifications it takes the form of confidence that all sequences of process interactions are understood when a few have been tried; in maintenance there is the hope that a small change can be certified by a test. For the design and coding phases, an effective specification serves as oracle, and when execution produces correct results in a few cases, we want to conclude that the software is correct.

Reliability [28]

Practical testing is an error-finding process. A good test uncovers something wrong. At the level of a programming-language procedure, it is possible to identify "error-prone" constructions, but an attempt to improve languages by eliminating these seems more an exercise in psychology than mathematics. Error-discovery testing is an art, but computer tools that perform bookkeeping can be invaluable aids in its practice. (Has this statement been tried? Could this expression be off by 1? Is this assertion true each time through the loop? Are all the variables assigned values before they are used? Etc.) Eventually a time comes when no more errors are discovered; at this point the question arises whether this is only an accident. Formally, is a successful test reliable [29], i.e., does it guarantee correctness?

A number of concessions have already been made when the question is asked in this form. To know if a test is a success requires an effective specification. It is assumed that the test is completed--that is, the program halts so that its output can be judged. In practice both of these things may cause a person considerable trouble in obtaining a successful test. Finally, the reliable tests are not to be generated, but only recognized; a person must find the test values. Despite these concessions, the problem of recognizing a reliable test is unsolvable. It is instructive to prove this result in a somewhat peculiar way.

Definition: A test T for program P is a finite set whose members are suitable inputs for P . T is successful iff P halts on each member as input, with the correct output. T is reliable for P iff

T successful ==> P correct.

Lemma: A reliable test exists for each program and specification.

Proof: Suppose P is incorrect. Then there is an input x for which the result does not agree with the specification, and {x} is reliable because it is not successful, falsifying the antecedent of the definition. On the other hand, if P is correct, the empty test is reliable, because the consequent of the definition is true.

Lemma: The problem of deciding whether or not an arbitrary program halts for no input (the empty-function equivalence problem) is unsolvable.

Proof. Suppose to the contrary that there were an effective way to decide if programs never halted. Then the usual halting problem could be solved. Given an arbitrary program P and input x , modify P so that it always appears to take x as input. (In most languages this can be done by replacing a READ-statement with an appropriate assignment.) Then P halted on x iff the modified program is not equivalent to one computing the empty function.

Lemma: The problem of deciding of an arbitrary program P , and its arbitrary statement C , whether or not C can be executed, (the useless-statement problem) is unsolvable.

Proof. Suppose there were such an algorithm. Then it could be used to solve the empty-function equivalence problem as follows. Given any program P , obtain a modified P' by adding a procedure that calls P , followed by any statement C . Then P computes the empty function just in case C is never executed in P' .

Theorem: The problem of recognizing reliable tests is unsolvable.

Proof. Suppose to the contrary that there were an algorithm to decide of an arbitrary T and P whether or not T is reliable for P . Then the useless-statement problem could be solved as follows. Suppose given an arbitrary P and its statement C . Construct a new program Q by adding a new variable (say N) to P , with N initially 0. Replace C by

```
C;
N := 1
```

and alter the expression E for the output of P to E+N . With these changes Q will agree with P exactly should C not be executed; otherwise P and Q will not agree. (Some assumptions about the form of P have been made, but appropriate changes in the construction can be made if these are violated.) Now take the input-output behavior of P to be the specification for Q , and find a reliable test T by trial, since one exists by the first lemma. If some value in T causes C to be executed, then the same would have been true for P , and hence C is not useless. On the other hand, if C remains unexecuted, P and Q agree on T , hence T is successful, and because it is reliable, Q is correct. That is, P and Q agree everywhere. But that means that C is never executed.

Faced with a proof of this kind, computer-science theory has an option not available in most other disciplines: it can deny hard facts. Indeed the useless-statement problem is unsolvable, but since there is no compelling reason to have useless statements in programs, we can insist that there be none. And such a requirement can be checked as a part of testing. Suppose then that the definition of "success" is changed to require that in addition to halting and correct results, every statement is executed in the course of a successful test. (Such a requirement can be checked by simple monitoring of the test executions.) The burden of dealing with undecidability has been placed back on a human being. With the modified definition, the proof that reliability is undecidable fails, since the supposed reliability algorithm need apply only to those programs without useless statements.

An absence of useless statements is neither necessary nor sufficient for test reliability, but they represent an anomaly probably not intended by the programmer. Calling attention to an apparently useless statement is thus a helpful bookkeeping device, and should aid in error-discovery testing. (The error is most likely that the test data is inadequate, and the statement not in fact useless.) Similarly, other variants of the undecidable-reliability proof can be invalidated by insisting on extra test requirements, which can be incorporated into useful testing tools. The success of these tools rests on the skill of the programmer using them, however, not on reliability.

The unsolvable problem that really underlies undecidable reliability is that of program equivalence. It is unreasonable to alter this hard fact of programming life, because the whole process of subdividing and combining programs relies on the language properties that lead to an unrecognizable collection of programs with the same meaning. It may be, however, that attempts to control the form of equivalent programs might have important consequences for testing.

"Automatic programming" is an idea closely allied with decidable reliability. Instead of beginning with a program and looking for reliable data, an automatic programming system begins with data and constructs a program [30]. That is, automatic programming is an algorithm for transforming a finite support, for a specification function f , into a program that computes f . Structural requirements on the programs permitted are essential to confidence that a synthesized program might be what was wanted; for example, to create programs with useless statements in response to a collection of test data is obviously silly—what should the content of those statements be? An effective reliability algorithm could be used as an automatic programming system by generating programs at random, and testing each against the given finite-support data for reliability. If a program were found for which this data is reliable, it would be acceptable.

Test Selection

Test selection methods are all based on dividing the input domain into equivalence classes defined by program and specification structure. The idea is that since both of these formal objects are finite, there should be a finite number of classes, and if inputs in each class are "treated the same" by the program ("should be treated the same" according to the specification), then choosing one element from each intersection of program and specification classes will yield a reliable test [31, 32]. It is unclear how to define "treated the same" so as to make this statement provable. For programs and operational specifications, some natural equivalence classes exist, however.

Path Equivalence classes [33] are the most natural for programs. Two inputs are in the same class iff they cause the program to take the same execution path, a finite sequence of statements from beginning to termination. An operational specification without process communication has similar path equivalence classes, but the intersection of program and specification classes may refine both. There are two difficulties with path equivalence classes: (1) loops and recursive procedure calls may give rise to an infinity of paths, hence an infinite subdivision of the domain, and (2) arbitrary choice of an element from each class may not yield a reliable test. There is no good solution to the first problem except to combine all but a finite number of loop- or recursion-originated path classes, perhaps using some intuition about which are "the same treatment." A rule of thumb is to group paths into classes which can be defined by a simple pattern. The intuitive reason behind difficulty (2) is that inputs that cause execution of the same path are not necessarily treated the same--as a simple example, for a path on which the formula

$$X/Y$$

appears, a zero-value for Y is given unique treatment.

To attack this second intuitive deficiency in path equivalence classes, computation equivalence classes [34] are defined on a path. These classes are much less natural than the path classes, but some are:

All inputs that lead to the same output. (Not useful when the path computation has an infinite range, but the path often precludes this.)

All inputs that lead to the same execution history. (These are literally treated the same, but the number of inputs in each class tends to 1.)

All inputs which can detect alterations in a given statement, expression, variable, etc., in that the result with the alteration would be different. The intuition behind this idea is that alterations might correct errors in the program; data in the class detects such errors.

Non-operational specifications may have natural equivalence classes of inputs as well. For example, a logic specification given as a conjunction of conditional assertions has the classes defined by individually satisfying each conjunct's hypothesis. An algebraic-axiom specification whose representation mapping is many-one has input classes whose members map to the same (canonical) word. However, the usefulness of such specification classes depends on properties of each example, not on the general form as in path and computation classes for programs.

To define input equivalence classes like those in the single-program case for cooperating processes requires supplying the inter-process communication. The practical expression of this situation arises in the testing of a concurrent system. Following "unit test" of individual processes with communication information supplied at random, the entire system is tried in an "integration test" where communication is allowed to happen naturally. It is observed that initial integration tests fail so dramatically that it is very difficult to discover the reasons; that is, unit tests are not very useful. A scheme for improving their utility might be the following. (We assume a slightly more realistic model of parallel computation in which the communication streams are directed to specific processes.) Unit tests are conducted in stages. At stage 0, each process is tested using communication inputs generated at random, and recording the communication outputs. At stage N, the communication inputs are those produced as outputs in stage N-1, supplied in a time sequence that makes it equally likely for communication to cause the program to wait as to allow it to proceed immediately.

Classes of interest over an entire multi-process program or operational specification certainly include those in which the computation histories within processes are fragmented and interleaved in different ways, yet produce the same output. If those histories are in addition required to be the same, members of the class are being treated the same up to an irrelevant scheduling. (An operating system is required to behave this way with the user processes it schedules.) A weaker class (among many) would be those inputs that cause the same sequence of process activations, without regard for the computations within them. Practical intuition on which to base natural equivalence classes is weak for cooperating processes, because disciplined testing of such systems is seldom attempted.

In the myriad of equivalence classes that can be defined by the several specification techniques, and the similar breakdown of program inputs, the intuition persists that there must some of finite index for which selecting any element from each class yields a reliable test. There can be no effective, general solution to the unsolvable problem of reliability, but solutions of two kinds might be expected. First, an effective solution might be obtained by imposing restrictions on the form of programs and specifications. Since operational specifications of cooperating processes can best tolerate constraints, so they seem most promising for the first possibility. Second, a non-effective solution might be found for the general problem. Equivalence classes of finite index in the domain, but defined by undecidable semantic properties, might nevertheless be used in a new proof method. The proof of reliability would be done by hand, and data found in each class; then, successfully executing those tests would prove the program correct.

Two views of testing theory arise from the unsolvability of the effective reliability problem. The first is that testing theory should provide new program proof techniques in which the computer is helpful, but the character of the human effort is similar to that of any proof: the necessary ideas are found in a non-algorithmic way [35]. This view has been presented above. In the second view, the required theoretical work should be done at a meta level, so that its results can be incorporated into testing tools for automatic application. The use of such tools is like using a compiler for syntax checking. Syntax theory is incorporated in a compiler in such a way that its users need only read and deal with very specific error messages, and these do not refer to grammar or parsing theory, yet that theory was needed to devise the compiler.

An ideal testing tool would have the following character. Its inputs would be a program, a test, and an effective specification. Its output would be either a message that the program is correct, or syntax-error-like messages faulting the program or test or both for failure to be reliable. Existing test tools have something of this character, except that they use no specifications, and they cannot produce the "program correct" message. For example, a statement-frequency analyzer identifies never-used statements, which clearly point to a deficiency in either the data or the program, and give the clues necessary to repair them. Existing test tools thus behave ideally when there are errors, but not when they detect none. There might be three ways to do better, each requiring theoretical development: (1) restrict the programming language to one for which the program-equivalence problem is solvable, thus making reliability effective; (2) weaken the requirements for an ideal test to establish something less than correctness, yet more than agreement with specification on the test points; (3) treat the test as an experiment to predict the outcome of other experiments with a certain degree of confidence. Each of these ideas will now be considered; (2) and (3) appear more promising.

Programming-language Restrictions

Finite-state transducers have a well-developed reliability theory of exactly the sort we seek for programs: given a bound on the state size, a test always exists, and can be effectively generated, to distinguish any machine from any other [36]. Such a "determining" test is therefore reliable, because only the determined machine can be correct. The state bound is not a problem in practice, since one has a program that is probably close to correct, and a rough bound can be taken to be some multiple of its size. Unfortunately, finite-state-like languages are not much use in practice, but before we consider that difficulty, the determining result can be studied to learn about reliability.

Suppose a test were reliable, in that a unique finite state transducer is successful, within a given state bound. That test, and the finite collection of results to which it leads with the machine it determines, constitutes a specification, since given the test and results, there is no further need for an oracle. This is a peculiar situation at odds with intuition. We have a finite set of input-output pairs that somehow define an infinite set, but do not describe that infinite set. The best description that can be given is to determine the machine, and let it (or some form like a regular expression constructed from it) be the specification off the reliable test. It is quite possible that the original specification is at odds with the determined machine for some untested points, because the required program is not finite state. Adding any such pair to the test will (perhaps with other additional points) determine a different machine (perhaps within a larger state bound).

What is interesting in this notion is how the original specification disappears, and this would happen with any reliable test. Behavior on the test collection fixes the behavior everywhere, but does not describe it except with a program. The reliable-test specification has the character: "get these particular results (finite list); then elsewhere do whatever you must." A more satisfactory specification that disagrees with this one might arise not because one of the restrictions is violated, but because of a subtle error in the reliable test. A mistake there will determine the wrong machine, but it may show its wrongness much more clearly on some untested point. (This situation is common in practical program testing, where a reported bug can often be traced to a test that was failed, but the failure missed or ignored.)

In a deeper sense, finite specifications of the sort established by reliable tests may be all that human beings are capable of mastering. That is, the most natural way of specifying most problems may be to constrain the programs for the solution, then give an adequate sample of the required behavior. Most automatic programming systems work in exactly that way, except that the class of programs they use is less precise than the finite-state class. They generate programs in a certain form which is a subset of some language, in response to a finite collection of inputs. Extending the collection generally results in a different program being generated up to some point, and after that the program does not change. It is of course easy to construct examples in which this behavior is misleading, but it is difficult to escape the feeling that in most cases these specifications by example and restricted program are very natural. The difficulty is that when we are misled, the joke is played out to the end: there is no fault in the process of obtaining a program, but it is wrong nevertheless. People seem more comfortable with a difficult process whose failures can be traced to faulty use, than with processes whose perfect application nevertheless goes wrong.

Finite-state devices are in one sense the ultimate in so-called "applicative" programming, in that they have no associated data storage. The common programming device of performing a calculation, then saving the result in a variable to be used later, after another intervening calculation that is unrelated, cannot be used. (So long as the values to be saved constitute a finite list, a machine exists to perform an equivalent computation, but in the most unnatural way of making a path choice based on the value that should have been stored, and on each path duplicating the intervening computation up to the test point, where the paths begin to differ. The same situation arises in any applicative language if the "compute-save-compute-retrieve" pattern is followed.) The deficiencies of finite-state programming are more serious, however, because of the tight coupling between the number of states and the memory. This restriction is one of practice as well as a theoretical limitation, because not only are some tasks impossible, but approximation to them causes the needed machine to grow exponentially in state size. (Parenthetically, this suggests a way to detect that automatic programming is failing: increase the size of the input, and if the generated program continually grows, it is a good guess that the answers being supplied are wrong. For example, asked to recognize balanced parentheses, the finite-state machine determined by finite tests would never stabilize, since there is no general way to do the problem.)

There is no natural candidate for a language significantly extending finite-state computing power, for which the reliability problem is solvable. However, languages with primitive-recursive power [37] have been analyzed with more success than universal languages: although the program-equivalence problem is unsolvable, programs always halt.

Between Correctness and Testing

The unsolvable program-equivalence problem is the reason that that reliable tests cannot be recognized. Even in universal programming languages, unsolvable questions can often be decided if "computation behavior" is substituted for input-output behavior. A program's computation is usually defined by operational semantics, along the lines used by Turing. A comprehensive program state is defined, and pieces of program syntax authorize transitions from one state to another. The sequence of state descriptions beginning with an initial one that includes the input value and ending with a final one that includes the output value, each leading to the next according to the program, is the computation (on that input with that output). The computation function for a program has the graph given by all such (input, computation) pairs.

(For Turing machines the Kleene normal-form theorem shows that computations can be recognized using a subrecursive function; this result is typical of the difference between input-output and input-computation behavior: the set

$$\{(p, x, y) \mid \text{program } p \text{ on input } x \text{ has output } y\}$$

is not recursive; but

$$\{(p, x, t) \mid \text{program } p \text{ on input } x \text{ has computation } t\}$$

is primitive recursive.)

Consider then "computational reliability," in which a program's specification is a computation function, and "correct" means having that computation function. Then some reliable tests are recognizable for some programs.

Theorem: There is a universal class of programs C such that each $P \in C$ has a nonempty subset of its determining tests T_P, for which

$$R = \{(P, T) \mid T \in T_P \text{ and } P \in C\}$$

is recursive.

Proof. The idea of the proof is that a canonical form of any program can be reconstructed from certain finite samples of its computations. The decision procedure required must determine whether T and the computations by P on T is such a sample. Details of the proof depend on the programming language used, but we outline the case of a language with assignments, conditionals, and loops, with expressions involving a fixed set of operators. First, any expression in P is determined by a finite collection of computations, those sufficient to distinguish all the operators from possible alternatives. It must, however, be possible to execute each expression with the necessary inputs. Assignment statements, conditionals, and loops are then distinguished by the computational actions following their expression evaluation. For conditionals it must be possible to reach both alternatives, and for loops there must be a case of zero iterations, and of more than one iteration, to distinguish a loop from a conditional. These caveats define the class of programs C and tests T_P for programs $P \in C$, which is evidently not restricted enough that any computable function is omitted.

The restrictions on programs and tests in the theorem all have the character of "no useless statements," which can be precisely reported when a program and test are being processed. For example, if statements of the program are not executed by the test (or executed with too little data to determine their expressions), the test might still be reliable (if the statements cannot be so executed at all) but the program and test are not in the recognizable subset of the theorem. A computer testing system can then produce specific error messages (e.g., "This expression has not been distinguished from the following alternatives: ...") and a person must decide if the test is inadequate, or if the program should be simplified because the unexplored situation cannot occur.

The idea of "computation" in the theorem can be replaced by other, weaker ideas at the cost of further restricting the sets of programs and tests that can be identified as reliable. At some point in the process of moving from the defining computation to mere input-output behavior, the set of programs ceases to be universal, which is another aspect of the undecidable reliability problem. The remarks made about "determining" tests for finite-state machines apply to computationally reliable tests as well: the test replaces the full specification, and the off-test behavior is only implied through the program thereby determined.

Probabilistic Testing

Quality control for manufactured objects consists of sampling the production, testing the sample, and inferring the likelihood that any given item will be faulty from the fraction of faulty samples. The success of this procedure depends on a number of things: (1) Each item is produced by the same process, and the introduction of a fault is as likely in one as in another; (2) The testing is exhaustive, in that no unit can pass the test, yet fail in actual use; (3) The number of samples is large enough. The best analogy in software testing is that each test of a program is a sample of its behavior, and the likelihood of failure in use can be inferred from the fraction of test failures. It is evidently true that confidence to be placed in results still depends on the number of tests; but parallels for assumptions (1) and (2) are more difficult to find.

Faults in software do not seem the same as manufacturing flaws because they are hidden in code in peculiar ways. Although it is an oversimplification, suppose a program's faults have fixed textual locations, with a uniform (or perhaps Poisson) distribution. (The basis for this assumption is that errors arise from loss of intellectual control of program complexity, which is as likely to occur at one place in the code as another. Certainly such a distribution is the right assumption for typographical errors.) Further suppose that the liklihood of detecting a fault is proportional to the number of test cases that reach its textual location in execution, with distinct program states. Test inputs do not explore a program's text uniformly, so even this very simple model shows that the probability of correctness based on a simple black-box model [38] are erroneous.

However, probabilistic ideas do offer promise of explaining why it is beneficial to exercise or cover a program. On the simple model that errors have a textual distribution, a better estimate of correctness can be based on how many distinct-path tests have passed through each control point. Since some program exercising is expensive to perform, it is also reasonable to examine the likelihood that when some randomly chosen fraction of coverage has been attained, that it will have exposed faults as effectively as full coverage [39].

When software is used, its inputs come from a distribution that is unlikely to be uniform over all possibilities. Hence a model of testing in which the source of program faults is not considered must draw tests from a similar distribution. The difficulty is that the operational data profile is not known for a new application.

Probabilistic theory, as an explanation for why practical test systems work as well as they do, as a source of new test methods, and as a fundamental explanatory model of the software testing process, has barely begun to be used.

References

1. B. W. Boehm, Software engineering, IEEE Trans. Computers C-25 (Dec., 1976), 1226-1241.

2. K. L. Heninger, Specifying software requirements for complex systems: new techniques and their application, IEEE Trans. Software Engineering SE-6 (Jan., 1980), 2-13.

3. B. H. Liskov and V. Barzins, An appraisal of program specifications, in P. Wegner et al., eds., Research Directions in Software Technology, MIT Press, 1979, 277-301.

4. W. P. Stevens, G. F. Myers, and L. C. Constantine, Structured design, IBM Systems J. 13 (1974), 115-139.

5. S. H. Caine and E. K. Gordon, PDL--a tool for software design, Proc. NCC 1975, 271-276.

6. G. J. Myers, The Art of Software Testing, Wiley, 1979.

7. R. L. Glass and R. A. Noiseux, Software Maintenance Guidebook, Prentice-Hall, 1981.

8. P. Naur, ed., Revised report on the algorithmic language ALGOL 60, CACM 6 (Jan., 1963), 1-17.

9. F. G. Pagan, Formal Specification of Programming Languages, Prentice-Hall, 1981.

10. A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison Wesley, 1977.

11. 2nd ACM SIGSOFT Software Engineering Symposium, Workshop on rapid prototyping, Columbia, MD., 1982.

12. D. L. Parnas, A technique for software module specification with examples, CACM 15 (March, 1972),330-336.

13. L. Robinson and O. Roubine, SPECIAL--a specification and assertion language, TR CSL-46, Stanford Research Institute, 1977.

14. B. J. Walker, R. A. Kemmerer, and G. J. Popek, Specification and verification of the UCLA UNIX security kernel, CACM 23 (Feb., 1980), 118-131.

15. K. L. Clark, Negation as failure, in H. Gallaire and J. Minker, eds., Logic and Data Bases, Plenum Press, 1978, 293-324.

16. J. Guttag and J. Horning, The algebraic specification of abstract data types, Acta Informatica 10 (1978), 27-52.

17. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, Abstraction mechanisms in CLU, CACM 20 (Aug., 1977), 564-576.

18. E. Mendelson, Introduction to Mathematical Logic, Van Nostrand, 1964 (Section 5.3)

19. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, An initial algebra approach to the specification, correctness, and implementation of abstract data types, in R. Yeh, ed., Current Trends in Programming Methodology, vol. IV,

Prentice-Hall, 1978, 80-149.

20. S. Kamin, Final data type specifications: a new data type specification method, Proc. 7th ACM Symposium on Principles of Programming Languages, Las Vegas, 1980, 131-138.

21. D. Musser, Abstract data type specification in the AFFIRM system, Proc. Specification for Reliable Software, Boston, 1979, 47-57.

22. J. A. Goguen and J. J. Tardo, An introduction to OBJ: a language for writing and testing formal algebraic programming specifications, Proc. Specification for Reliable Software, Boston, 1979, 170-189.

23. P. Zave, The operational approach to requirements specification for embedded systems, Univ. of Md. TR-976, December, 1980.

24. C. A. R. Hoare, Communicating sequential processes, CACM 21 (Aug., 1978), 666-677.

25. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

26. W. E. Riddle et al., Behavior modelling during software design, IEEE Trans. Software Engineering SE-4 (July, 1978), 283-298.

27. G. Kahn, The semantics of a simple language for parallel processing, IFIP 74, Stockholm, 471-475.

28. R. G. Hamlet, Reliability theory of program testing, Acta Informatica 16 (1981), 31-43.

29. W. E. Howden, Reliability of the path analysis strategy, IEEE Trans. Software Engineering SE-2 (Sept., 1976), 208-215.

30. P. D. Summers, A methodology for LISP program construction from examples, JACM 24 (Jan., 1977), 161-175.

31. J. B. Goodenough and S. L. Gerhart, Towards a theory of test data selection, IEEE Trans. Software Engineering SE-1 (June, 1975), 156-173.

32. E. J. Weyuker and T. J. Ostrand, Theories of program testing and the application of revealing subdomains, IEEE Trans. Software Engineering SE-7 (May 1980), 236-246.

33. L. J. White and E. I. Cohen, A domain strategy for computer program testing, IEEE Trans. Software Engineering SE-7 (May 1980), 247-257.

34. R. G. Hamlet, Testing programs with finite sets of data, The Computer J. 20 (1977), 232-237.

35. E. J. Weyuker, Assessing test data adequacy through program inference, to appear in TOPLAS.

36. Z. Kohavi et al., Machine distinguishing experiments, The Computer J. 16 (May, 1973), 141-147.

37. A. R. Meyer and D. M. Ritchie, The complexity of LOOP programs, Proc. 22nd ACM National Conf., 1967, 465-469.

38. J. W. Duran and J. J. Wiorkowski, Toward models for probabilistic program correctness, Proc. Software Quality and Assurance Workshop, San Diego, 1978,

39-44.

39. S. Pheha, A software quality assurance methodology for air force projects, MITRE Corp., 1981.